Course - 2 Date: 25/07/2024

WEEK 1: Hyperparameter Tuning, Batch Normalisation, Programming frameworks

Practical aspects of Deep Learning

Applied ML is highly iterative process

~~# layers~~

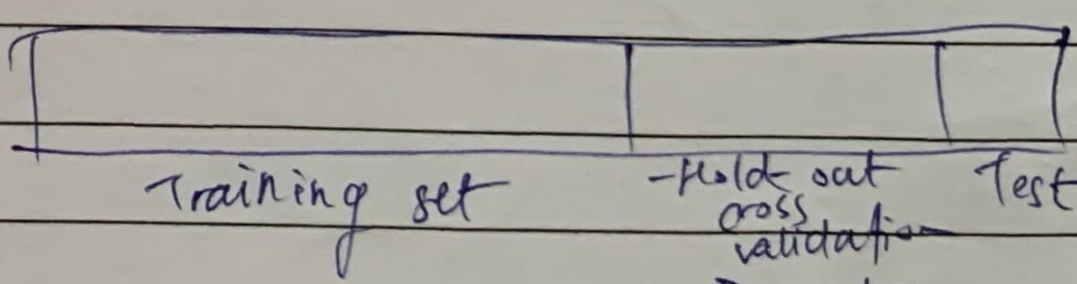
~~# hidden units~~

Learning rates

activation functions

NLP, CV, Speech recg., Structured Data

→ Train / dev / Test sets



Train

Date:

Previous era: 70 – 30 ← Test

60 – 20 – 20 ← Small data set

↓ train ↓ dev ↓ test

Big data era:

[devsets just chooses best choices of different algorithms, which is working best]

98 – 1 – 1 ← Big dataset

99.5 – 0.25 – 0.25

99.5 – 0.4 – 0.1

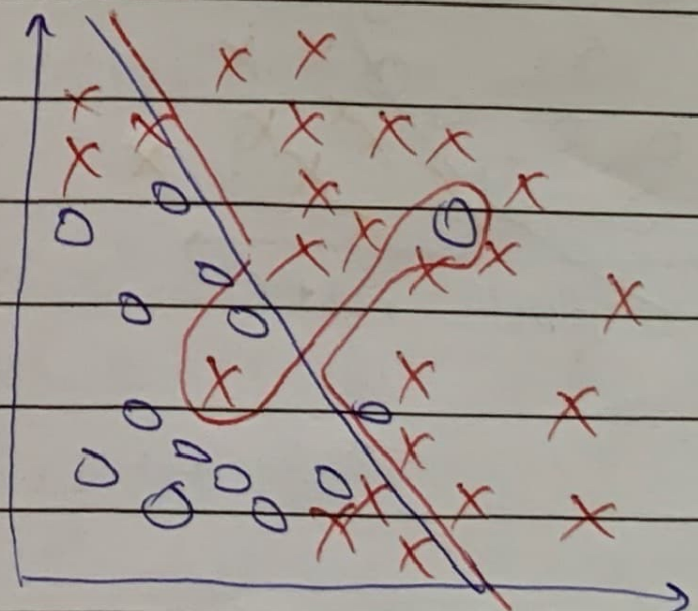
Date:

→ Mismatched train/test distribution :

Make sure the dev and test set come from the same distribution.

→ Not having a test set might be okay.
∵ Test set gives you the unbiased estimate of how your final network works.

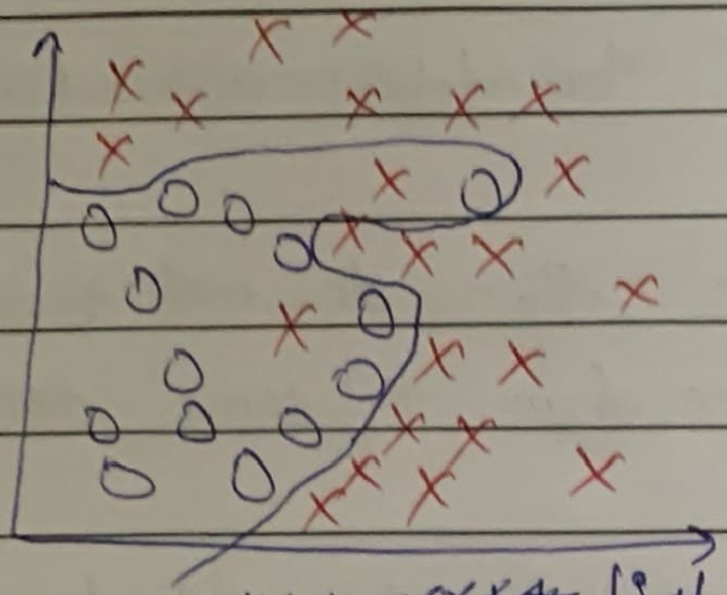
→ Bias and Variance



{Note: The red plot boundary is high bias & high variance
∵ it is mostly linear and middle two mislabels are covered}

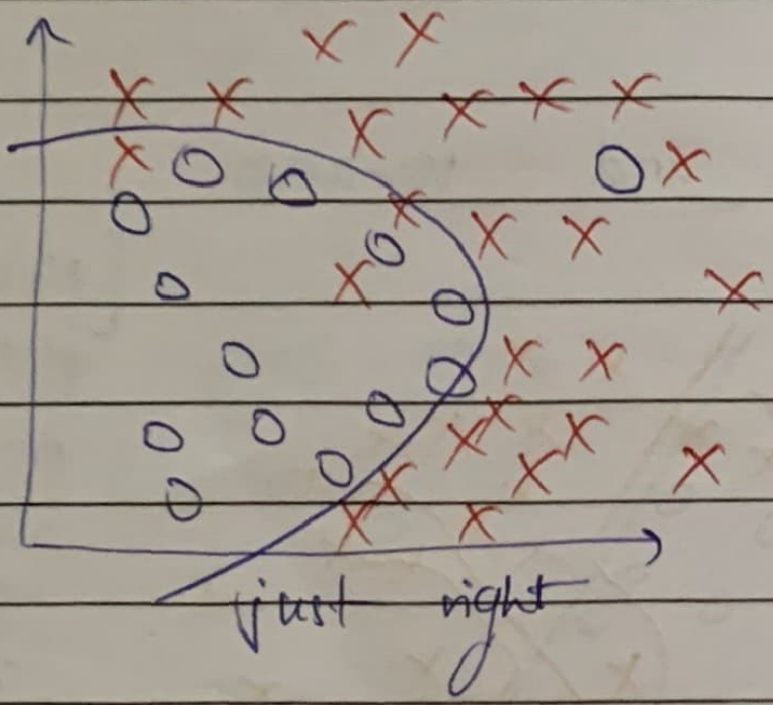
high bias
" underfitting "

Date:



~~just right~~ high variance

"overfitting"



Date:

Example:

→ Cat classification:

Train set error:	1%	15%
Dev set error:	11%	16%
	high variance	high bias

Human ≈ 0%.

Optimal (Bayes) error ≈ 0%	15%	0.5%
	30%	1%
	high bias high variance	low bias low variance

The analysis depends on the optimal error.

→ Basic "recipe" for ML

High Bias? (training data perform.)

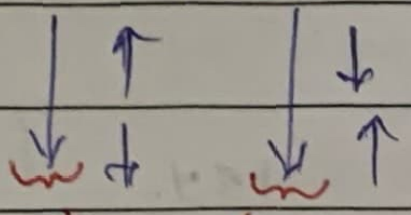
Y → ① Bigger network ② Train longer ③ (NN architecture search)

N ↓

Date:

High variance? → ① more data
(dev set perform.) ↓ ② Regularisation
↓ M ③ NN architecture selection
Done

"Bias Variance trade-off"



Deep Learning to decrease both

"Training a bigger NN almost never hurts."

Date:

⇒ Regularization: (helps reducing high variance)

Logistic Regression

$w \in \mathbb{R}^{n_x}$; $b \in \mathbb{R}$

$$\min_{w,b} J(w,b)$$

$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

$$+ \frac{\lambda}{2m} \|w\|_2^2$$

regularization parameter (hyperparameter)

L_2 regularization (most common type):

$$\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$

{Note: $+ \frac{\lambda}{2m} b^2$ can also be done, but not used ∵ b is just a number w has all the features in it}

{Note: we will write "lambd" in our code instead of "lambda". }

Date:

L1 regularization

$$\frac{\lambda}{m}\sum_{j=1}^{n_x}|w_j| = \frac{\lambda}{2m}||w||_1$$

Here, w will be sparse.

NN:

$$J(w^{[1]}, b^{[1]}, \cdots w^{[L]}, b^{[L]})$$

$$= \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}||w^{[l]}||_F^2$$

$$||w||_F^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}(w_{ij}^{[l]})^2 \qquad w: (n^{[l]}, n^{[l-1]})$$

"Frobenius norm"

Date:

$dw \rightarrow$ (from back prop.) $+ \frac{\lambda}{m} w^{[l]}$

$w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$

(can be called as :
"weight decay" :: $w^{[l]} * \left(1 - \frac{\lambda\alpha}{m}\right) \approx < 1$)

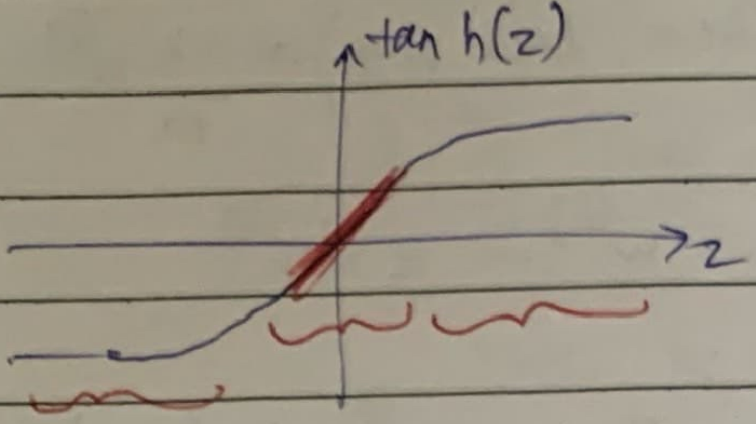
$w^{[l]} := w^{[l]} - \alpha \left[(\text{from back prop}) + \frac{\lambda}{m} w^{[l]} \right]$

$\therefore w^{[l]} := w^{[l]} - \frac{\alpha \lambda w^{[l]}}{m}$

$- \alpha (\text{from backprop})$

If the value of lambda is set to "very high" then the weight matrix $w^{[l]} \approx 0$
and high variance turns to high bias.
this causes zeroing or reducing impact of hidden units. The many of hidden units don't make an impact and hence the Deep NN converts to a smaller NN.

Date: 26/07/2024



$\lambda \uparrow \quad w^{[l]} \downarrow \quad z^{[l]} : w^{[l]} a^{[l-1]} + b^{[l]}$

$z^{[l]} \downarrow \quad \therefore g^{[l]}(z^{[l]})$ will roughly linear since the tan h(z) at small z values is almost linear.

Date:

→ Dropout Regularisation:

For each training example, the hidden units are diminished with a given propability and then ~~the~~ it is trained.

Implementing Dropout ("Inverted Dropout")

Illustrate in layer $l = 3$;

$d_3 = np.random.rand(a_3.shape[0], a_3.shape[1]) < keep\_prob$

keep-prob = 0.8 — 0.2 chance of eliminating any hidden unit.

$a_3 = np.multiply(a_3, d_3)$ # $a_3 *= d_3$

Date:

$a_3 /= keep\text{-}prob$ Note: This step ensures that value of a_3 is still same so that there is no scaling problem.

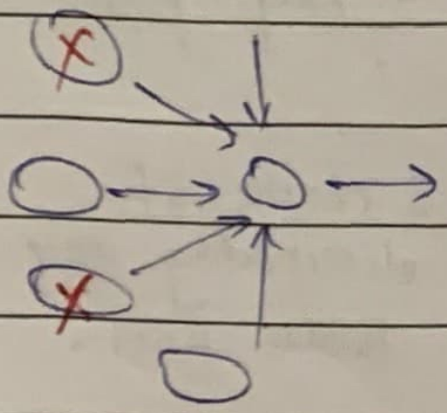
Making predictions at test time:

No drop-out.

Why does drop-out work?

→ Can't rely on any one feature, so have to spread weights.

⇝ Shrink weights



Drop-out and L_2 regularisation have almost the same effect.

Date:

You could keep the value of keep-prob different for different layers, depending on the size of the weight matrix.

For a ~~higher~~ big weight matrix; keep-prob could be 0.5

otherwise keep-prob can be probably higher than 0.5.

The calculation of cost function $J(w,b)$ becomes difficult along with the drop out.

⇒ Regularization methods:

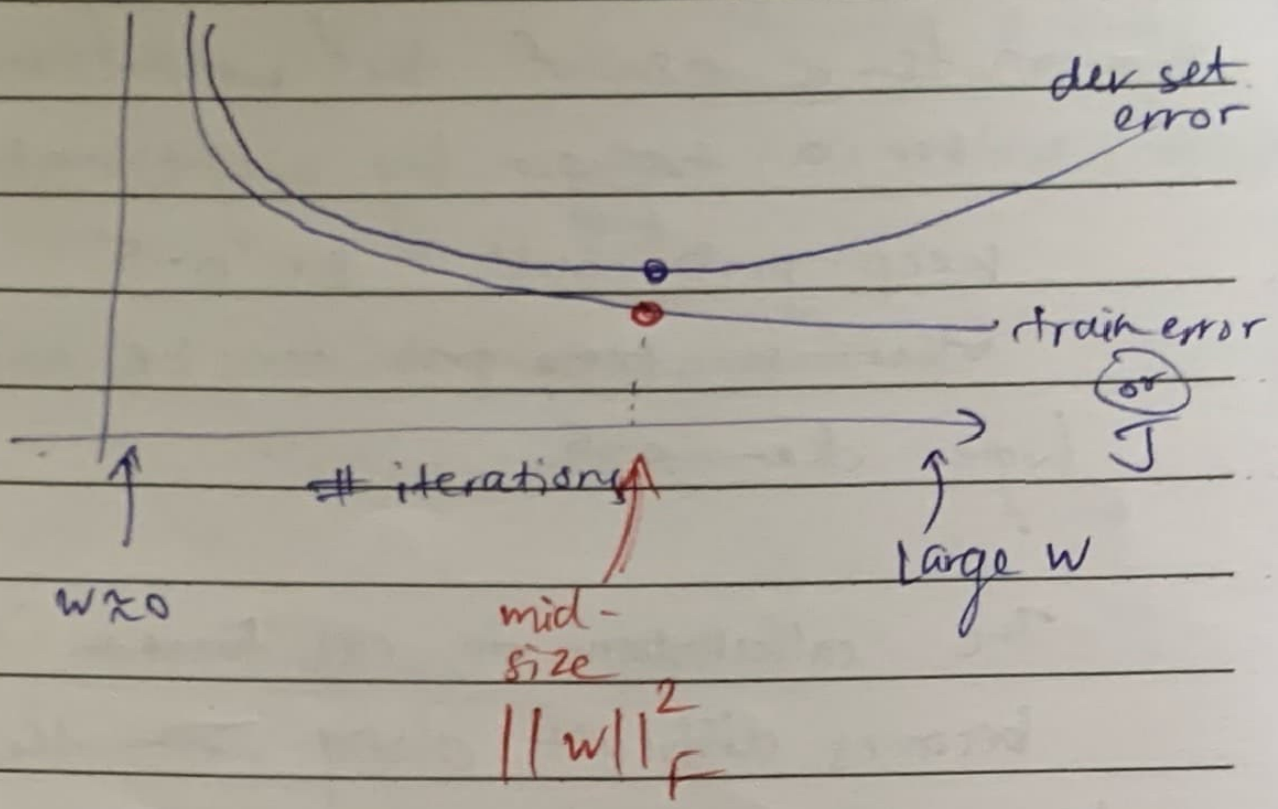
→ Data augmentation:

Example: To prevent overfitting, in a cat image classification, flip the images horizontally, which will get 2m training examples.

Thus, can get more data by augmenting the images.

Date:

→ Early Stopping (stops the training ~~early~~ early)



Early stopping carries out two processes together called as orthogonalisation.

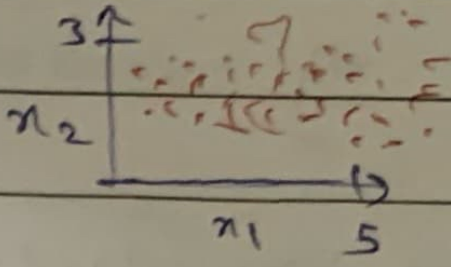
1. Optimise cost fn J
2. Not overfit

hence to stop early it will even stop the minimising of cost function J.

Date:

Normalizing training sets:

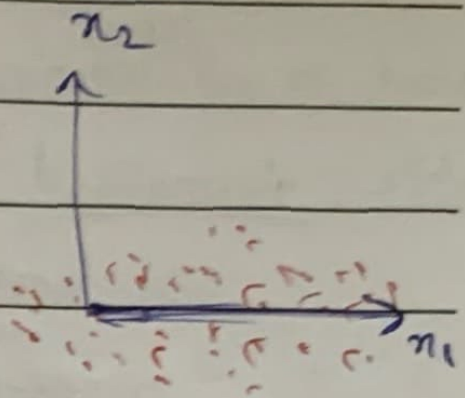
$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$



Steps:

(1) $$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$$

$$x := x - \mu$$



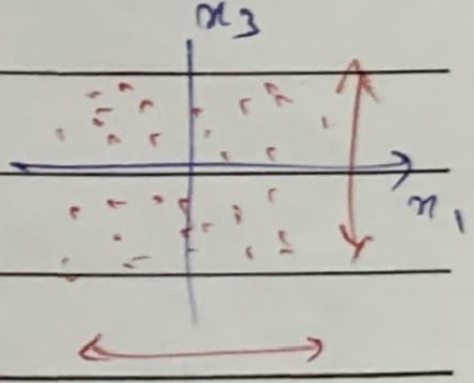
(2) Normalize variance

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \ast\ast 2$$

$\ast\ast 2$ → element-wise

vector of variances of each of the features

$$x /= \sigma$$

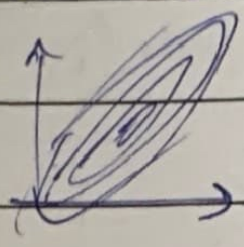


Date:

Use same μ, σ to normalise test-set.

Why normalize inputs?

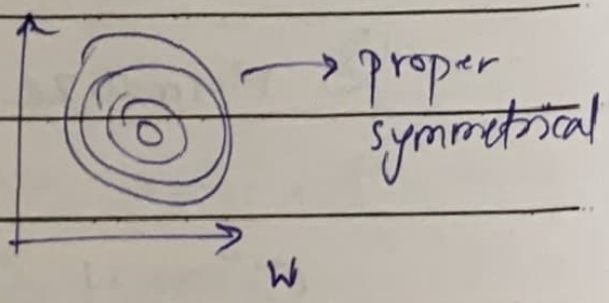
While Unnormalised: The cost function is very squished



Need to use small learning rate

While normalised:

cost function on average will look symmetric.

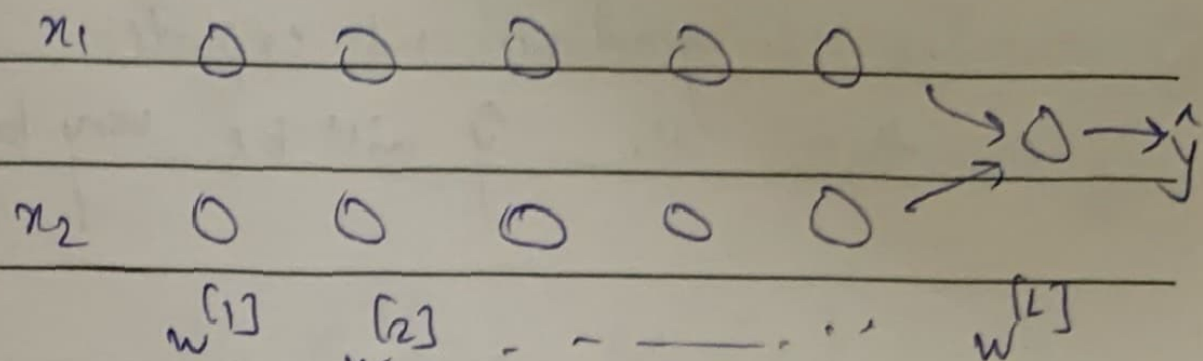


can be use ↑ learning rate.

→ Easier to optimise when normalised.

Date:

Vanishing / Exploding gradients:



$g(z) = z$, $b^{[l]} = 0$

$$\hat{y} = w^{[L]} \cdot w^{[L-1]} \cdot w^{[L-2]} \cdots w^{[2]} w^{[1]} X$$

$$z^{[1]} = w^{[1]} x$$

$$a^{[1]} = g(z^{[1]}) = z^{[1]}$$

$$a^{[2]} = g[z^2] = g(w^{[2]} a^{[1]})$$

$$W^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

$$\hat{y} = W^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} X$$

~~$\hat{y} = W^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} X$~~

$$\hat{y} = W^{[L]} 1.5^{L-1} X$$

Date:

∴ If → Deep NN very deep

L is very large

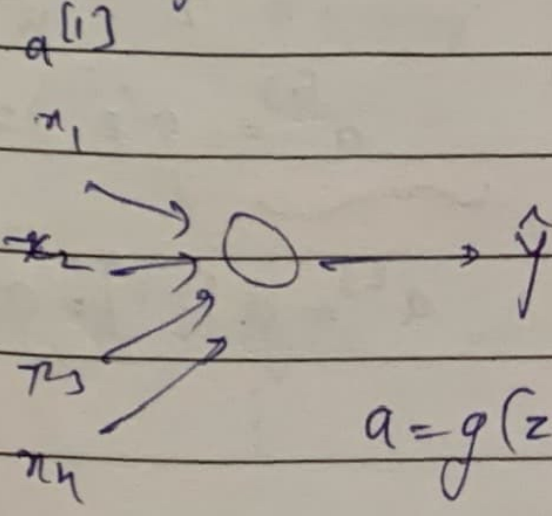
then the gradients will explode;

then the $\hat{y}$ will be very large.

$W^{[l]} > I$ (explode)

$W^{[l]} < I$ (activations decrease [vanishing exponentially])

→ Weight initialisation for DNN:



$a = g(z)$

$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$

large n → small w_i

Date:

$$Var(w_i) = \frac{1}{n}$$

$$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$$

If $g^{[l]}(z) = ReLU(z)$

then $Var(w_i) = \frac{2}{n}$

and $np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$

Other variants:

for tanh(): $np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$

→ Xavier Initialisation

other version ↗

$$np.sqrt\left(\frac{2}{n^{[l-1]} + n^{[l]}}\right)$$

Date:

Gradient check for a NN:

Take $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$ ---

and reshape into

big vector θ

$J = (w^{[1]}, b^{[1]} - \cdots, w^{[L]}, b^{[L]}) = J(\theta)$

Take $dw^{[1]}, db^{[1]} \cdots dw^{[L]}, db^{[L]}$

and reshape into a

big vector $d\theta$

Gradient checking (Grad check):

$J(\theta) = J(\theta_1, \theta_2 \cdots)$

for each i:

$$d\theta_{approx}^{[i]} = \frac{J(\theta_1, \theta_2 \cdots, \theta_i + \varepsilon \cdots) - J(\theta_1, \theta_2 \cdots, \theta_i - \varepsilon \cdots)}{2\varepsilon}$$

[i] Date:

$d\theta[i] = \frac{\partial J}{\partial \theta_i}$

$d\theta_{approx} \approx d\theta$

Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$ ← Euclidean lengths

→ Euclidean distance

This is calculating Euclidean distance

formula : $\sqrt{\sum_{i=1}^{n} (d\theta_{approx} - d\theta)^2}$

$\varepsilon = 10^{-7}$ if ans = 10^{-7} → great! (gradients)

if ans = 10^{-5} → double-check

if ans = 10^{-3} → worry (bug)

$> 10^{-3}$ → worry.

derivatives → might be wrong

Date:

→ Gradient checking Implementation Notes :

(1) Don't use while training ;
use only for debug

(2) If algorithm fails grad check,
look at components to try
to identify bug. Check for ($db^{[l]}, dw^{[l]}$)

(3) Remember regularization.

(4) Doesn't work with dropout

(5) Run at random initialisation, & check
if the it work values of $w^{[l]}, b^{[l]}$
$\neq 0$.